# THE LINUX PROGRAMMING INTERFACE

A Linux and UNIX® System Programming Handbook

MICHAEL KERRISK

no starch press

# 43

# INTERPROCESS COMMUNICATION OVERVIEW

This chapter presents a brief overview of the facilities that processes and threads can use to communicate with one another and to synchronize their actions. The following chapters provide more details about these facilities.

## 43.1 A Taxonomy of IPC Facilities

Figure 43-1 summarizes the rich variety of UNIX communication and synchronization facilities, dividing them into three broad functional categories:

- *Communication*: These facilities are concerned with exchanging data between processes.
- *Synchronization*: These facilities are concerned with synchronizing the actions of processes or threads.
- *Signals*: Although signals are intended primarily for other purposes, they can be used as a synchronization technique in certain circumstances. More rarely, signals can be used as a communication technique: the signal number itself is a form of information, and realtime signals can be accompanied by associated data (an integer or a pointer). Signals are described in detail in Chapters 20 to 22.

Although some of these facilities are concerned with synchronization, the general term *interprocess communication* (IPC) is often used to describe them all.

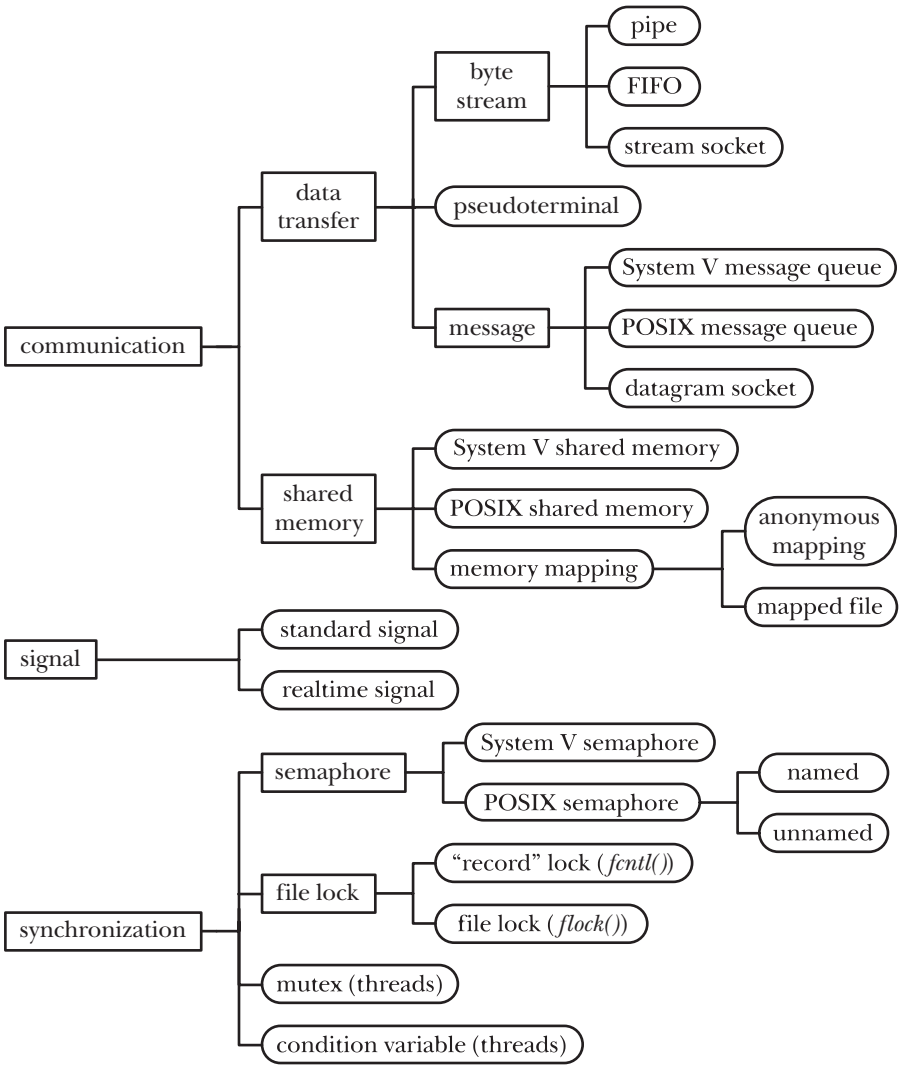


**Figure 43-1:** A taxonomy of UNIX IPC facilities

As Figure 43-1 illustrates, often several facilities provide similar IPC functionality. There are a couple of reasons for this:

- Similar facilities evolved on different UNIX variants, and later came to be ported to other UNIX systems. For example, FIFOs were developed on System V, while (stream) sockets were developed on BSD.
- New facilities have been developed to address design deficiencies in similar earlier facilities. For example, the POSIX IPC facilities (message queues, semaphores, and shared memory) were designed as an improvement on the older System V IPC facilities.

In some cases, facilities that are grouped together in Figure 43-1 actually provide significantly different functionality. For example, stream sockets can be used to communicate over a network, while FIFOs can be used only for communication between processes on the same machine.

## 43.2 Communication Facilities

The various communication facilities shown in Figure 43-1 allow processes to exchange data with one another. (These facilities can also be used to exchange data between the threads of a single process, but this is seldom necessary, since threads can exchange information via shared global variables.)

We can break the communication facilities into two categories:

- *Data-transfer facilities*: The key factor distinguishing these facilities is the notion of writing and reading. In order to communicate, one process writes data to the IPC facility, and another process reads the data. These facilities require two data transfers between user memory and kernel memory: one transfer from user memory to kernel memory during writing, and another transfer from kernel memory to user memory during reading. (Figure 43-2 shows this situation for a pipe.)
- *Shared memory*: Shared memory allows processes to exchange information by placing it in a region of memory that is shared between the processes. (The kernel accomplishes this by making page-table entries in each process point to the same pages of RAM, as shown in Figure 49-2, on page 1026.) A process can make data available to other processes by placing it in the shared memory region. Because communication doesn't require system calls or data transfer between user memory and kernel memory, shared memory can provide very fast communication.

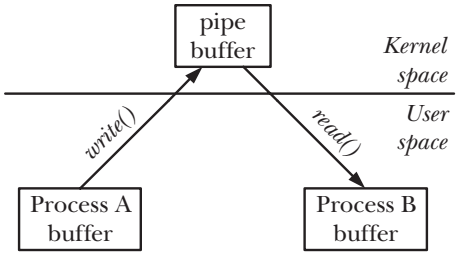


**Figure 43-2:** Exchanging data between two processes using a pipe

### Data transfer

We can further break data-transfer facilities into the following subcategories:

- *Byte stream*: The data exchanged via pipes, FIFOs, and datagram sockets is an undelimited byte stream. Each read operation may read an arbitrary number of bytes from the IPC facility, regardless of the size of blocks written by the writer. This model mirrors the traditional UNIX "file as a sequence of bytes" model.

- *Message*: The data exchanged via System V message queues, POSIX message queues, and datagram sockets takes the form of delimited messages. Each read operation reads a whole message, as written by the writer process. It is not possible to read part of a message, leaving the remainder on the IPC facility; nor is it possible to read multiple messages in a single read operation.
- *Pseudoterminals*: A pseudoterminal is a communication facility intended for use in specialized situations. We provide details in Chapter 64.

A few general features distinguish data-transfer facilities from shared memory:

- Although a data-transfer facility may have multiple readers, reads are destructive. A read operation consumes data, and that data is not available to any other process.

  > The `MSG_PEEK` flag can be used to perform a nondestructive read from a socket (Section 61.3). UDP (Internet domain datagram) sockets allow a single message to be broadcast or multicast to multiple recipients (Section 61.12).

- Synchronization between the reader and writer processes is automatic. If a reader attempts to fetch data from a data-transfer facility that currently has no data, then (by default) the read operation will block until some process writes data to the facility.

#### Shared memory

Most modern UNIX systems provide three flavors of shared memory: System V shared memory, POSIX shared memory, and memory mappings. We consider the differences between them when describing the facilities in later chapters (see Section 54.5 in particular).

Note the following general points about shared memory:

- Although shared memory provides fast communication, this speed advantage is offset by the need to synchronize operations on the shared memory. For example, one process should not attempt to access a data structure in the shared memory while another process is updating it. A semaphore is the usual synchronization method used with shared memory.
- Data placed in shared memory is visible to all of the processes that share that memory. (This contrasts with the destructive read semantics described above for data-transfer facilities.)

## 43.3 Synchronization Facilities

The synchronization facilities shown in Figure 43-1 allow processes to coordinate their actions. Synchronization allows processes to avoid doing things such as simultaneously updating a shared memory region or the same part of a file. Without synchronization, such simultaneous updates could cause an application to produce incorrect results.

UNIX systems provide the following synchronization facilities:

- *Semaphores*: A semaphore is a kernel-maintained integer whose value is never permitted to fall below 0. A process can decrease or increase the value of a semaphore. If an attempt is made to decrease the value of the semaphore below 0, then the kernel blocks the operation until the semaphore's value increases to a level that permits the operation to be performed. (Alternatively, the process can request a nonblocking operation; then, instead of blocking, the kernel causes the operation to return immediately with an error indicating that the operation can't be performed immediately.) The meaning of a semaphore is determined by the application. A process decrements a semaphore (from, say, 1 to 0) in order to reserve exclusive access to some shared resource, and after completing work on the resource, increments the semaphore so that the shared resource is released for use by some other process. The use of a binary semaphore—a semaphore whose value is limited to 0 or 1—is common. However, an application that deals with multiple instances of a shared resource would employ a semaphore whose maximum value equals the number of shared resources. Linux provides both System V semaphores and POSIX semaphores, which have essentially similar functionality.
- *File locks*: File locks are a synchronization method explicitly designed to coordinate the actions of multiple processes operating on the same file. They can also be used to coordinate access to other shared resources. File locks come in two flavors: read (shared) locks and write (exclusive) locks. Any number of processes can hold a read lock on the same file (or region of a file). However, when one process holds a write lock on a file (or file region), other processes are prevented from holding either read or write locks on that file (or file region). Linux provides file-locking facilities via the *flock()* and *fcntl()* system calls. The *flock()* system call provides a simple locking mechanism, allowing processes to place a shared or an exclusive lock on an entire file. Because of its limited functionality, *flock()* locking facility is rarely used nowadays. The *fcntl()* system call provides record locking, allowing processes to place multiple read and write locks on different regions of the same file.
- *Mutexes and condition variables*: These synchronization facilities are normally used with POSIX threads, as described in Chapter 30.

  Some UNIX implementations, including Linux systems with a *glibc* that provides the NPTL threading implementation, also allow mutexes and condition variables to be shared between processes. SUSv3 permits, but doesn't require, an implementation to support process-shared mutexes and condition variables. They are not available on all UNIX systems, and so are not commonly employed for process synchronization.

When performing interprocess synchronization, our choice of facility is typically determined by the functional requirements. When coordinating access to a file, file record locking is usually the best choice. Semaphores are often the better choice for coordinating access to other types of shared resource.

Communication facilities can also be used for synchronization. For example, in Section 44.3, we show how a pipe can be used to synchronize the actions of a parent process with its children. More generally, any of the data-transfer facilities can be

used for synchronization, with the synchronization operation taking the form of exchanging messages via the facility.

> Since kernel 2.6.22, Linux provides an additional, nonstandard synchronization mechanism via the *eventfd()* system call. This system call creates an *eventfd* object that has an associated 8-byte unsigned integer maintained by the kernel. The system call returns a file descriptor that refers to the object. Writing an integer to this file descriptor adds that integer to the object's value. A *read()* from the file descriptor blocks if the object's value is 0. If the object has a nonzero value, a *read()* returns that value and resets it to 0. In addition, *poll()*, *select()*, or *epoll* can be used to test if the object has a nonzero value; if it does, the file descriptor indicates as being readable. An application that wishes to use an *eventfd* object for synchronization must first create the object using *eventfd()*, and then call *fork()* to create related processes that inherit file descriptors referring to the object. For further details, see the *eventfd(2)* manual page.

## 43.4 Comparing IPC Facilities

When it comes to IPC, we face a range of choices that can at first seem bewildering. In later chapters that describe each IPC facility, we include sections that compare each facility against other similar facilities. In the following pages, we consider a number of general points that may determine the choice of IPC facility.

### IPC object identification and handles for open objects

In order to access an IPC object, a process must have some means of identifying the object, and once the object has been "opened," the process must use some type of handle to refer to the open object. Table 43-1 summarizes these properties for the various types of IPC facilities.

**Table 43-1:** Identifiers and handles for various types of IPC facilities

| Facility type | Name used to identify object | Handle used to refer to object in programs |
|---|---|---|
| Pipe | no name | file descriptor |
| FIFO | pathname | file descriptor |
| UNIX domain socket | pathname | file descriptor |
| Internet domain socket | IP address + port number | file descriptor |
| System V message queue | System V IPC key | System V IPC identifier |
| System V semaphore | System V IPC key | System V IPC identifier |
| System V shared memory | System V IPC key | System V IPC identifier |
| POSIX message queue | POSIX IPC pathname | *mqd_t* (message queue descriptor) |
| POSIX named semaphore | POSIX IPC pathname | *sem_t ** (semaphore pointer) |
| POSIX unnamed semaphore | no name | *sem_t ** (semaphore pointer) |
| POSIX shared memory | POSIX IPC pathname | file descriptor |
| Anonymous mapping | no name | none |
| Memory-mapped file | pathname | file descriptor |
| *flock()* lock | pathname | file descriptor |
| *fcntl()* lock | pathname | file descriptor |

### Functionality

There are functional differences between the various IPC facilities that can be relevant in determining which facility to use. We begin by summarizing the differences between data-transfer facilities and shared memory:

- Data-transfer facilities involve read and write operations, with transferred data being consumable by just one reader process. Flow control between writer and reader, as well as synchronization (so that a reader is blocked when trying to read data from a facility that is currently empty) is automatically handled by the kernel. This model fits well with many application designs.
- Other application designs more naturally suit a shared-memory model. Shared memory allows one process to make data visible to any number of other processes sharing the same memory region. Communication "operations" are simple—a process can access data in shared memory in the same manner as it accesses any other memory in its virtual address space. On the other hand, the need to handle synchronization (and perhaps flow control) can add to the complexity of a shared-memory design. This model fits well with application designs that need to maintain shared state (e.g., a shared data structure).

With respect to the various data-transfer facilities, the following points are worth noting:

- Some data-transfer facilities transfer data as a byte stream (pipes, FIFOs, and stream sockets); others are message-oriented (message queues and datagram sockets). Which approach is preferable depends on the application. (An application can also impose a message-oriented model on a byte-stream facility, by using delimiter characters, fixed-length messages, or message headers that encode the length of the total message; see Section 44.8.)
- A distinctive feature of System V and POSIX message queues, compared with other data-transfer facilities, is the ability to assign a numeric type or priority to a message, so that messages can be delivered in a different order from that in which they were sent.
- Pipes, FIFOs, and sockets are implemented using file descriptors. These IPC facilities all support a range of alternative I/O models that we describe in Chapter 63: I/O multiplexing (the *select()* and *poll()* system calls), signal-driven I/O, and the Linux-specific *epoll* API. The primary benefit of these techniques is that they allow an application to simultaneously monitor multiple file descriptors to see whether I/O is possible on any of them. By contrast, System V message queues don't employ file descriptors and don't support these techniques.

  > On Linux, POSIX message queues are also implemented using file descriptors and support the alternative I/O techniques described above. However, this behavior is not specified in SUSv3, and is not supported on most other implementations.

- POSIX message queues provide a notification facility that can send a signal to a process, or instantiate a new thread, when a message arrives on a previously empty queue.

- UNIX domain sockets provide a feature that allows a file descriptor to be passed from one process to another. This allows one process to open a file and make it available to another process that otherwise might not be able to access the file. We briefly describe this feature in Section 61.13.3.
- UDP (Internet domain datagram) sockets allow a sender to broadcast or multicast a message to multiple recipients. We briefly describe this feature in Section 61.12.

With respect to process-synchronization facilities, the following points are worth noting:

- Record locks placed using *fcntl()* are considered to be owned by the process placing the lock. The kernel uses this ownership property to detect deadlocks (situations where two or more processes are holding locks that block each other's further lock requests). If a deadlock situation occurs, the kernel denies the lock request of one of the processes, returning an error from the *fcntl()* call to indicate that a deadlock occurred. System V and POSIX semaphores don't have an ownership property; no deadlock detection occurs for semaphores.
- Record locks placed using *fcntl()* are automatically released when the process that owns the locks terminates. System V semaphores provide a similar feature in the form of an "undo" feature, but this feature is not reliable in all circumstances (Section 47.8). POSIX semaphores don't provide an analog of this feature.

#### Network communication

Of all of the IPC methods shown in Figure 43-1, only sockets permit processes to communicate over a network. Sockets are generally used in one of two domains: the UNIX domain, which allows communication between processes on the same system, and the *Internet* domain, which allows communication between processes on different hosts connected via a TCP/IP network. Often, only minor changes are required to convert a program that uses UNIX domain sockets into one that uses Internet domain sockets, so an application that is built using UNIX domain sockets can be made network-capable with relatively little effort.

#### Portability

Modern UNIX implementations support most of the IPC facilities shown in Figure 43-1. However, the POSIX IPC facilities (message queues, semaphores, and shared memory) are not quite as widely available as their System V IPC counterparts, especially on older UNIX systems. (An implementation of POSIX message queues and full support for POSIX semaphores have appeared on Linux only in the 2.6.*x* kernel series.) Therefore, from a portability point of view, System V IPC may be preferable to POSIX IPC.

#### System V IPC design issues

The System V IPC facilities were designed independently of the traditional UNIX I/O model, and consequently suffer a few peculiarities that make their programming

interfaces more complicated to use. The corresponding POSIX IPC facilities were designed to address these problems. The following points are of particular note:

- The System V IPC facilities are connectionless. These facilities provide no notion of a handle (like a file descriptor) referring to an open IPC object. In later chapters, we'll sometimes talk of "opening" a System V IPC object, but this is really just shorthand to describe the process of obtaining a handle to refer to the object. The kernel does not record the process as having "opened" the object (unlike other types of IPC objects). This means that the kernel can't maintain a reference count of the number of processes that are currently using an object. Consequently, it can require additional programming effort for an application to be able to know when an object can safely be deleted.
- The programming interfaces for the System V IPC facilities are inconsistent with the traditional UNIX I/O model (they use integer key values and IPC identifiers instead of pathnames and file descriptors). The programming interfaces are also overly complex. This last point applies particularly to System V semaphores (refer to Sections 47.11 and 53.5).

By contrast, the kernel counts open references for POSIX IPC objects. This simplifies decisions about when an object can be deleted. Furthermore, the POSIX IPC facilities provide an interface that is simpler and more consistent with the traditional UNIX model.

### Accessibility

The second column of Table 43-2 summarizes an important characteristic of each type of IPC object: the permissions scheme that governs which processes can access the object. The following list adds some details on the various schemes:

- For some IPC facilities (e.g., FIFOs and sockets), object names live in the file system, and accessibility is determined according to the associated file permissions mask, which specifies permissions for owner, group, and other (Section 15.4). Although System V IPC objects don't reside in the file system, each object has an associated permissions mask whose semantics are similar to those for files.
- A few IPC facilities (pipes, anonymous memory mappings) are marked as being accessible only by related processes. Here, *related* means related via *fork()*. In order for two processes to access the object, one of them must create the object and then call *fork()*. As a consequence of the *fork()*, the child process inherits a handle referring to the object, allowing both processes to share the object.
- The accessibility of a POSIX unnamed semaphore is determined by the accessibility of the shared memory region containing the semaphore.
- In order to place a lock on a file, a process must have a file descriptor referring to the file (i.e., in practice, it must have permission to open the file).
- There are no restrictions on accessing (i.e., connecting or sending a datagram to) an Internet domain socket. If necessary, access control must be implemented within the application.

**Table 43-2:** Accessibility and persistence for various types of IPC facilities

| Facility type | Accessibility | Persistence |
|---|---|---|
| Pipe | only by related processes | process |
| FIFO | permissions mask | process |
| UNIX domain socket | permissions mask | process |
| Internet domain socket | by any process | process |
| System V message queue | permissions mask | kernel |
| System V semaphore | permissions mask | kernel |
| System V shared memory | permissions mask | kernel |
| POSIX message queue | permissions mask | kernel |
| POSIX named semaphore | permissions mask | kernel |
| POSIX unnamed semaphore | permissions of underlying memory | depends |
| POSIX shared memory | permissions mask | kernel |
| Anonymous mapping | only by related processes | process |
| Memory-mapped file | permissions mask | file system |
| *flock()* file lock | *open()* of file | process |
| *fcntl()* file lock | *open()* of file | process |

### Persistence

The term *persistence* refers to the lifetime of an IPC object. (Refer to the third column of Table 43-2.) We can distinguish three types of persistence:

- *Process persistence*: A process-persistent IPC object remains in existence only as long as it is held open by at least one process. If the object is closed by all processes, then all kernel resources associated with the object are freed, and any unread data is destroyed. Pipes, FIFOs, and sockets are examples of IPC facilities with process persistence.

  > The persistence of a FIFO's data is not the same as the persistence of its name. A FIFO has a name in the file system that persists even after all file descriptors referring to the FIFO have been closed.

- *Kernel persistence*: A kernel-persistent IPC object exists until either it is explicitly deleted or the system is shut down. The lifetime of the object is independent of whether any process holds the object open. This means that, for example, one process can create an object, write data to it, and then close it (or terminate). At a later point, another process can open the object and read the data. Examples of facilities with kernel persistence are System V IPC and POSIX IPC. We exploit this property in the example programs that we present when describing these facilities in later chapters: for each facility, we implement separate programs that create an object, delete an object, and perform communication or synchronization.
- *File-system persistence*: An IPC object with file-system persistence retains its information even when the system is rebooted. The object exists until it is explicitly deleted. The only type of IPC object that demonstrates file-system persistence is shared memory based on a memory-mapped file.

### Performance

In some circumstances, different IPC facilities may show notable differences in performance. However, in later chapters, we generally refrain from making performance comparisons, for the following reasons:

- The performance of an IPC facility may not be a significant factor in the overall performance of an application, and it may not be the only factor in determining the choice of an IPC facility.
- The relative performance of the various IPC facilities may vary across UNIX implementations or between different versions of the Linux kernel.
- Most importantly, the performance of an IPC facility will vary depending on the precise manner and environment in which it is used. Relevant factors include the size of the data units exchanged in each IPC operation, the amount of unread data that may be outstanding on the IPC facility, whether or not a process context switch is required for each unit of data exchanged, and other load on the system.

If IPC performance is crucial, there is no substitute for application-specific benchmarks run under an environment that matches the target system. To this end, it may be worth writing an abstract software layer that hides details of the IPC facility from the application and then testing performance when different IPC facilities are substituted underneath the abstract layer.

## 43.5 Summary

In this chapter, we provided an overview of various facilities that processes (and threads) can use to communicate with one another and to synchronize their actions.

Among the communication facilities provided on Linux are pipes, FIFOs, sockets, message queues, and shared memory. Synchronization facilities provided on Linux include semaphores and file locks.

In many cases, we have a choice of several possible techniques for communication and synchronization when performing a given task. In the course of this chapter, we compared the different techniques in various ways, with the aim of highlighting some differences that may influence the choice of one technique over another.

In the following chapters, we go into each of the communication and synchronization facilities in much more detail.

## 43.6 Exercises

**43-1.** Write a program that measures the bandwidth provided by pipes. As command-line arguments, the program should accept the number of data blocks to be sent and the size of each data block. After creating a pipe, the program splits into two process: a child that writes the data blocks to the pipe as fast as possible, and a parent that reads the data blocks. After all data has been read, the parent should print the elapsed time required and the bandwidth (bytes transferred per second). Measure the bandwidth for different data block sizes.

**43-2.** Repeat the preceding exercise for System V message queues, POSIX message queues, UNIX domain stream sockets, and UNIX domain datagram sockets. Use these programs to compare the relative performance of the various IPC facilities on Linux. If you have access to other UNIX implementations, perform the same comparisons on those systems.